Weighted Block-Asynchronous Iteration on GPU-Accelerated Systems

Hartwig Anzt[1], Stanimire Tomov[2], Jack Dongarra[2,3,4], and Vincent Heuveline[1]

[1] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
[2] University of Tennessee, Knoxville, USA
[3] Oak Ridge National Laboratory, Oak Ridge, USA
[4] University of Manchester, Manchester, UK
{hartwig.anzt,vincent heuveline}@kit.edu,
{tomov,dongarra}@cs.utk.edu

Abstract. In this paper, we analyze the potential of using weights for block-asynchronous relaxation methods on GPUs. For this purpose, we introduce different weighting techniques similar to those applied in block-smoothers for multigrid methods. For test matrices taken from the University of Florida Matrix Collection we report the convergence behavior and the total runtime for the different techniques. Analyzing the results, we observe that using weights may accelerate the convergence rate of block-asynchronous iteration considerably. While component-wise relaxation methods are seldom directly applied to systems of linear equations, using them as smoother in a multigrid framework they often provide an important contribution to finite element solvers. Since the parallelization potential of the classical smoothers like SOR and Gauss-Seidel is usually very limited, replacing them by weighted block-asynchronous smoothers may be beneficial to the overall multigrid performance. Due to the increase of heterogeneity in today's architecture designs, the significance and the need for highly parallel asynchronous smoothers is expected to grow.

Keywords: asynchronous relaxation, weighted block-asynchronous iteration methods, multigrid smoother, GPU.

1 Introduction

Using weights in iterative relaxation schemes is a well known and often applied technique to improve the convergence. While the classical successive over-relaxation method (SOR, [Saa03]) consists of a weighted Gauss-Seidel, especially the block smoothers in multigrid methods are often enhanced with weights to improve their convergence [BFKMY11]. In these the parallelized Block-Jacobi- or Block-Gauss-Seidel smoothers are weighted according to the block decomposition of the matrix. In [ATG+11] we explored the potential of replacing the classically applied smoothers in multigrid methods by asynchronous iterations. While a block parallelized smoother requires synchronization between the iterations, asynchronous methods are very tolerant to component update order and

I. Caragiannis et al. (Eds.): Euro-Par 2012 Workshops, LNCS 7640, pp. 145–154, 2013.

latencies concerning the communication of updated component values. This lack of synchronization barriers makes them suitable candidates for modern hardware architectures, often accelerated by highly parallel coprocessors. In [ATDH11] we have shown how to enhance asynchronous iteration schemes to compensate for the inferior convergence rate of the plain asynchronous iteration. In particular, this is achieved by adding local iterations on subdomains that fit into the accelerators' cache, therefore almost come for free, and should not be counted as global iterations. Furthermore, the higher iteration number per time frame on the GPUs potentially results in significant performance increase. While Chazan and Miranker have introduced a weighted asynchronous iteration similar to SOR [CM69], it becomes of interest whether the block-asynchronous iteration benefits from weighting methods similar to those applied to block smoothers [BFKMY11]. The motivation is that in the local iterations performed on the subdomains, the off-block matrix entries are neglected. To account for this issue it may be beneficial to weight the local iterations. This can be achieved either by using ℓ_1-weights, by a technique similar to ω-weighting in SOR, or by a combination of both. The purpose of this paper is to introduce the different methods and report experimental results on the convergence rate as well as the time-to-solution performance.

2 Mathematical Background

2.1 Asynchronous Iteration

The Jacobi method is an iterative algorithm for finding the approximate solution to a linear system of equations $Ax = b$ that converges if A is strictly or irreducibly diagonally dominant [Var10], [Bag95]. One can decompose the system into $(L + D + U)x = b$ where D denotes the diagonal entries of A while L and U denote the lower and upper triangular part of A, respectively. Using the form $Dx = b - (L + U)x$, the Jacobi method is derived as an iterative scheme where the matrix $B \equiv I - D^{-1}A$ is often referred to as iteration matrix. It can also be rewritten in the following component-wise form:

$$x_i^{m+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j \neq i}^{n} a_{ij} x_j^m \right) = \sum_{i=1}^{n} B_{ij} x_j^m + d_i \tag{1}$$

where $b_{ij} = (B)_{ij}$ with $B = I - D^{-1}A$ and $d_i = \frac{b_i}{a_{ii}}$ for all $i, j = 1 \ldots n$. For computing the next iteration, one needs the latest values of all components. This requires a strict order of the component updates, limiting the parallelization potential to a stage, where no component can be updated multiple times before all the other components are updated. If this order is not adhered, i.e. the individual components are updated independently and without consideration of the current state of the other components, the resulting algorithm is called chaotic or asynchronous iteration method [FS00]. Back in the 70's Chazan and Miranker analyzed some basic properties of these methods, and established

convergence theory [CM69], [Str97] [AD86], [BE86]. For the last 30 years, these algorithms came out of focus of high-performance computing due to the superior performance of synchronized iteration methods. More interest was put on the convergence properties and deriving models for the computational cost [AD89], [Bah97], [BSS99], [DB91]. Today, due to the complexity of heterogeneous hardware platforms and the high number of computing units in parallel devices like GPUs, these schemes may become interesting again for applications like multigrid methods, where highly parallel smoothers are required on the distinct grid levels. While traditional smoothers like the sequential Gauss-Seidel obtain their efficiency from their fast convergence, it may be true that the asynchronous iteration scheme overcompensate the inferior convergence behavior by superior scalability [ATG+11].

2.2 Block-Asynchronous Iteration

One possible motivation for the block-asynchronous iteration comes from the hardware architecture. The idea is to split the linear system into blocks of rows, and then to assign the computations for each block to one thread block on a graphics processing unit (GPU). For these thread blocks, an asynchronous iteration method is used, while on each thread block, instead of one, multiple Jacobi-like iterations are performed. During these local iterations on the matrix block, the x values used from outside the block are kept constant (equal to their values at the beginning of the local iterations). After the local iterations, the updated values are communicated. In other words, using domain-decomposition terminology, the blocks correspond to subdomains and thus the method iterates locally on every subdomain. We denote this scheme by *async-(local_iters)*, where the index *local_iters* indicates the number of Jacobi-like updates on each subdomain [ATDH11]. As the subdomains are relatively small and the data needed for the local iterations largely fits into the multiprocessor's cache, these additional iterations on the subdomains almost come for free. This approach, inspired by the well know hybrid relaxation schemes [BFKMY11], [BFG+], arises as a specific case of an asynchronous two-stage method [BMPS99].

The obtained algorithm can be written as component-wise update of the solution approximation:

$$x_k^{(m+1)} += \frac{1}{a_{kk}} \left(\underbrace{b_k - \sum_{j=1}^{T_S-1} a_{kj} x_j^{(m-\nu(m+1,j))}}_{\text{global part}} \underbrace{- \sum_{j=T_S}^{T_E} a_{kj} x_j^{(m)}}_{\text{local part}} \underbrace{- \sum_{j=T_E+1}^{n} a_{kj} x_j^{(m-\nu(m+1,j))}}_{\text{global part}} \right)$$

where T_S and T_E denote the starting and the ending indices of the matrix/vector part in the thread block. In the component updates on the subdomains, for the local components, the most recent values are used, while for the global part, the values from the beginning of the iteration are used. The shift function $\nu(m+1, j)$ denotes the iteration shift for the component j which can be positive or negative,

depending on whether the thread block where the component j is located in already has conducted more or less iterations. Note that this may give a block Gauss-Seidel flavor to the updates.

2.3 Weights in Block-Asynchronous Iteration

To examine the topic of weights in the block-asynchronous iteration, we introduce some notation to simplify the analysis [BFKMY11]. Splitting the matrix A into blocks, we use A_{kk} for the matrix block located in the k-th block row and the k-th block column. Furthermore, we introduce the index sets Ω_k, where

$$\Omega = \bigcup_{k=1}^{p} \Omega_k = \{1, 2 \dots n\},$$

and $\Omega_i \cap \Omega_j = \emptyset \; \forall i \neq j$ consistent to the block decomposition of the matrix A. Using this notation, Ω_k contains all indices j with $T_S(k) \leq j \leq T_E(k)$ where $T_S(k)$ (respectively $T_E(k)$) denotes the first (last) row and column index of the diagonal block A_{kk}. We now define the sets

$$\Omega^{(i)} = \{j \in \Omega_k : i \in \Omega_k\}, \quad \Omega_0^{(i)} = \{j \notin \Omega_k : i \in \Omega_k\}.$$

Hence, for block A_{kk}, $\Omega^{(i)}$ contains the indices with corresponding columns being part of the diagonal block of row i while $\Omega_0^{(i)}$ contains the indices of the columns that have no entries in the block. This way, we can decompose the sum of the elements of the i-th row:

$$\sum_j a_{ij} = \underbrace{\sum_{j=1}^{T_S-1} a_{ij}}_{\text{off-block columns}} + \underbrace{\sum_{j=T_S}^{T_E} a_{ij}}_{\text{block columns}} + \underbrace{\sum_{j=T_E+1}^{n} a_{ij}}_{\text{off-block columns}} = \underbrace{\sum_{j \in \Omega^{(i)}} a_{ij}}_{\text{block columns}} + \underbrace{\sum_{j \in \Omega_0^{(i)}} a_{ij}}_{\text{off-block columns}}. \tag{2}$$

Similar to the ω-weighted asynchronous iteration [CM69], it is possible to use ω-weights for the block structure in the block-asynchronous approach. In this case, the solution approximation of the local iterations is weighted when updating the global iteration values. The parallel algorithm for the component updates in one matrix block is outlined in Algorithm 1.

Beside this general weighting method, we introduce a more sophisticated technique, that is usually referred to as ℓ_1-weighting [BFKMY11].

Classically applied to Block-Jacobi and Gauss-Seidel relaxation methods, ℓ_1 weighting modifies the iteration matrix by replacing $B = I - D^{-1}A$ with $B_{\ell_1} = I - (D + D^{\ell_1})^{-1}A$, where D^{ℓ_1} is the diagonal matrix with entries

$$d_{ii}^{\ell_1} := \mathbf{sign}(a_{ii}) \sum_{j \in \Omega_0^{(i)}} |a_{ij}|. \tag{3}$$

1: Update component i:
2: $s := d_i + \sum_{j \in \Omega_0^{(i)}} b_{ij} x_j$ {off-diagonal part}
3: $x^{local} = x$
4: **for all** $k = 0;\ k < local\_iters;\ k++$ **do**
5: $\quad x_i^{local} := s + \sum_{j \in \Omega^{(i)}} b_{ij} x_j^{local}$ {using the local updates in the block}
6: **end for**
7: $x_i = \omega x_i^{local} + (1 - \omega) x_i$

Algorithm 1. Basic principle of using ω weights in block-asynchronous iteration featuring *local_iters* local iterations

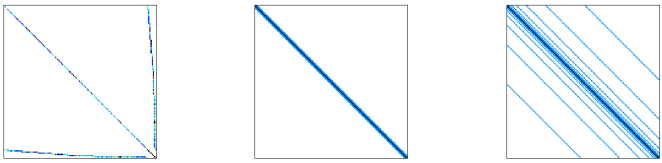
Table 1. Dimension and characteristics of the SPD test matrices and the corresponding iteration matrices

Matrix name	$\#n$	$\#nnz$	con(A)	con($D^{-1}A$)	$\rho(M)$
CHEM97ZTZ	2,541	7,361	1.3e+03	7.2e+03	0.7889
FV1	9,604	85,264	9.3e+04	12.76	0.8541
FV3	9,801	87,025	3.6e+07	4.4e+03	0.9993
TREFETHEN_2000	2,000	41,906	5.1e+04	6.1579	0.8601

The advantage over the across-the-board ω-weighting technique is that it applies different weights in the distinct rows, which accounts for the respective off-diagonal entries.

3 Numerical Experiments

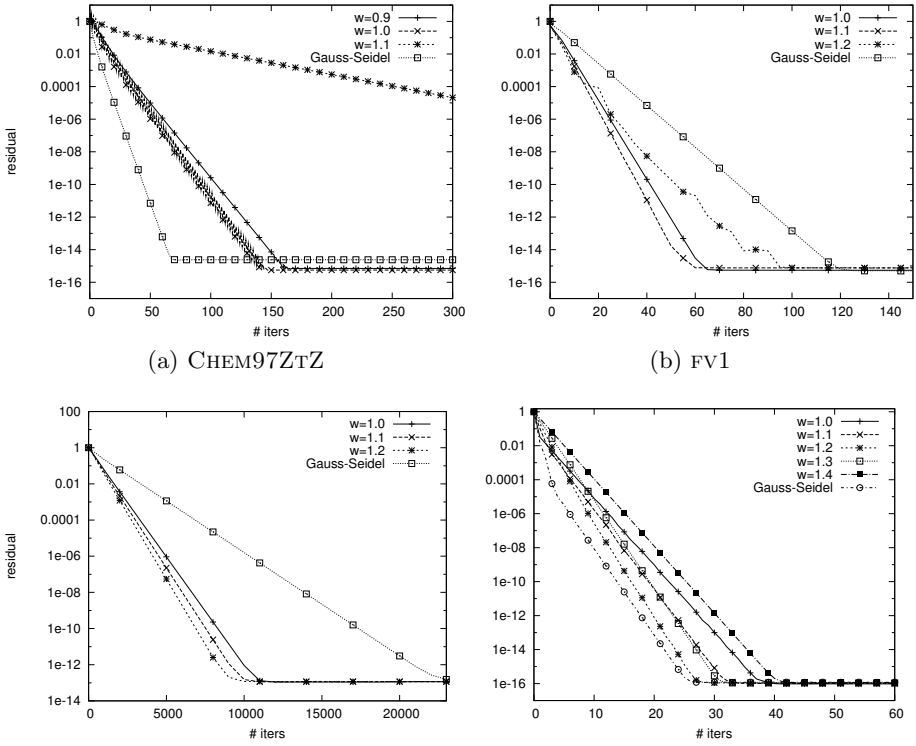
In our experiments, we search for the approximate solutions of linear system of equations, where the respective matrices are taken from the University of Florida Matrix Collection (UFMC; see `http://www.cise.ufl.edu/research/sparse/matrices/`). Due to the convergence properties of the iterative methods we analyze, the experiment matrices have to be chosen properly, fulfilling the sufficient convergence condition [Str97]. The matrix properties and sparsity plots can be found in Table 1 and Figure 1. We furthermore take set the right-hand side in $Ax = b$ to $b \equiv 1$ for all linear systems.



(a) CHEM97ZTZ (b) FV1, FV3 (c) TREFETHEN_2000

Fig. 1. Sparsity plots of test matrices

The experiments were conducted on a heterogeneous GPU-accelerated multicore system located at the University of Tennessee, Knoxville. The system's



(a) Chem97ZtZ (b) fv1

(c) fv3 (d) Trefethen_2000

Fig. 2. Convergence rate of ω-weighted block-asynchronous iteration compared to Gauss-Seidel convergence. The (relative) residuals are always in L^2 norm.

CPU is one socket Intel Core Quad Q9300 @ 2.50GHz and the GPU is a Fermi C2050 (14 Multiprocessors x 32 CUDA cores @1.15GHz, 3 GB memory). The GPU is connected to the CPU host through a PCI-e×16. On the CPU, the synchronous Gauss-Seidel and SOR implementations run on 4 cores. The Intel compiler 11.1.069 [int] is used with optimization flag "-O3". In the GPU implementation, based on CUDA [NVI09] with the respective libraries taken from CUDA 4.0.17 [NVI11], the component updates use a thread blocks of size 512. (Except for the ℓ_1 weighted method, where the thread block size is chosen consistent to the matrix block size.) The thread block size, the number of streams, along with other parameters, were determined through empirically based tuning.

In a first experiment, we analyze the influence of ω-weighting on the convergence rate with respect to global iteration numbers. For this purpose we plot the relative residual depending on the iteration number. Note that all values are average due to the non-deterministic properties of block-asynchronous iteration. To have a reference, we additionally provide in Figure 2 the convergence behavior of the sequential Gauss-Seidel algorithm. The results reveal that the convergence rate of the block-asynchronous iteration is very dependent on the

matrix characteristics. For matrices with most relevant matrix entries gathered on or near the diagonal, the local iterations provide sufficient improvement to compensate for the inferior convergence rate of the asynchronous iteration conducted globally. In these cases, e.g. FV1 and FV3, we achieve a higher convergence rate than the sequential Gauss-Seidel algorithm. Similar to the SOR algorithm, choosing $\omega > 1$ may even improve the convergence for specific problems (Figure 2b and 2c). For systems with considerable off-diagonal entries, the convergence of the block-asynchronous iteration decreases considerably compared to the Gauss-Seidel scheme (Figure 2a, 2d). The reason is, that the off-diagonal entries are ignored in the local iterations.

While the ω-technique applies a general weighting to account for the off-diagonal entries, the more sophisticated ℓ_1-weighting technique applies different weights in different rows. To analyze the impact of ℓ_1-weighting we focus on the matrix TREFETHEN_2000 where the ratio between the entries in the diagonal block and the off-diagonal parts differs significantly for the distinct rows.

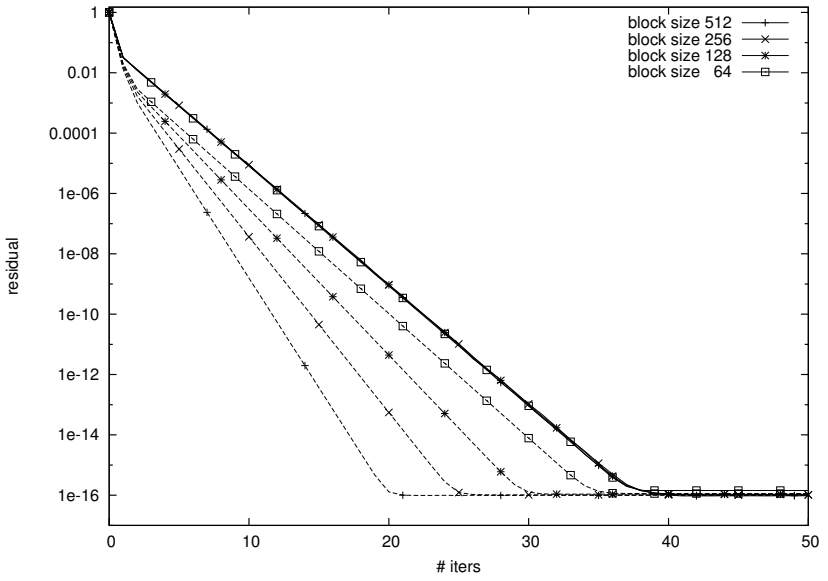
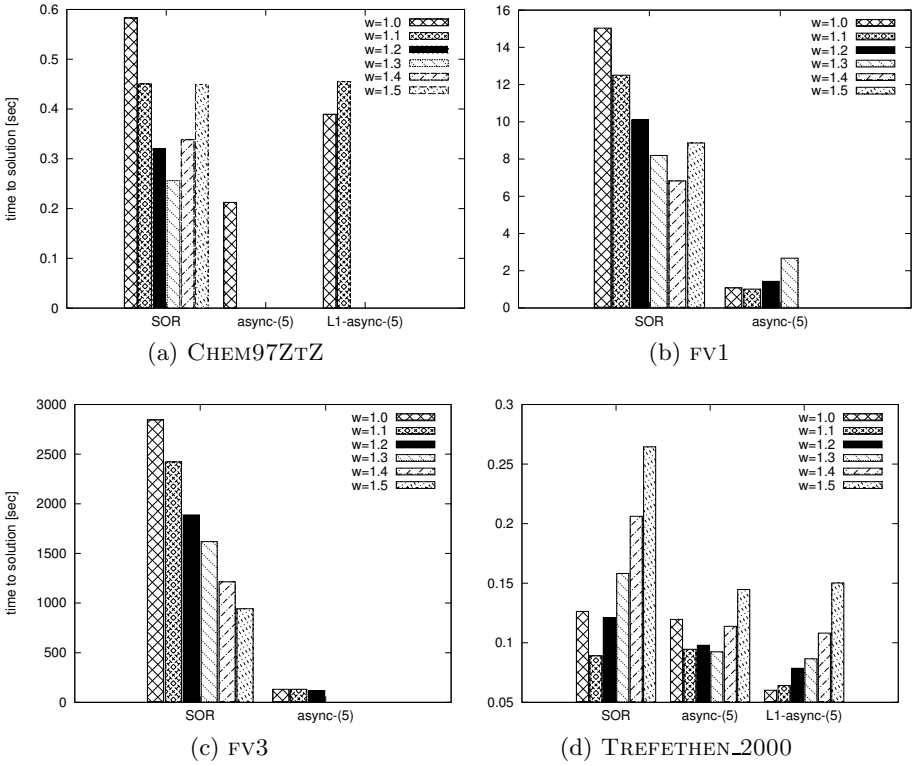


Fig. 3. Convergence improvement using ℓ_1-weights applied to TREFETHEN_2000 for different block sizes. Solid lines, all lying on top of each other, are block-asynchronous iteration, dashed lines are block-asynchronous iteration using ℓ_1-weights. The (relative) residuals are always in L^2 norm.

We can observe in Figure 3 that, independently of the block size, using ℓ_1 weights improves the convergence rate. We also note that the influence of the block-size on the convergence rate for the unweighted algorithm is negligible. Furthermore, using ℓ_1 weights is especially beneficial when targeting large block sizes, where the ℓ_1 weights for the distinct rows in one block differ considerably. For this case (e.g. block size 512), the convergence of the block-asynchronous



(a) Chem97ZtZ (b) fv1

(c) fv3 (d) Trefethen_2000

Fig. 4. Time-to-solution comparison between SOR and ω-weighted block-asynchronous iteration (async-(5)). In (4a) and (4d) we additionally provide the data for the combination of ω- and ℓ_1-weighting (L1-async-(5)).

iteration is improved by a factor of almost 2 compared to the unweighted algorithm.

While the convergence rate, with respect to iteration number, is interesting from the theoretical point of view, the more relevant factor is the time-to-solution performance. This depends not only on the convergence rate, but also on the efficiency of the respective algorithm on the available hardware resources (hardware-dependent iteration rate). Whereas the Gauss-Seidel algorithm and the derived SOR algorithms require strict update order and hence only allow sequential implementations, block-asynchronous iteration is very tolerant to update order and synchronization latencies, and therefore adequate for GPU implementations. In the next experiment, we analyze the time to solution for the ω-weighted block-asynchronous iteration and compare it with the SOR algorithm. We want to stress that despite the similar notation, ω-weighting has, due to the algorithm design, a very different meaning in the SOR and the block-asynchronous itera-

tion, respectively. While ω weights in SOR the individual iterations, in async-(5) it is used to weight the local iterations with respect to the global ones.

For the matrices with considerable off-diagonal entries (large d_{ii} in (3)), we provide additional data for different ω-weights applied to the block-asynchronous algorithm enhanced by the ℓ_1-weighting technique. The results show that for matrices where most entries are clustered on or near the main diagonal, the ω-weighted block asynchronous iteration outperforms the SOR method by more than an order of magnitude, see Figure 4b and 4c. But at the same time, ω-weights for the block-asynchronous algorithm have to be applied more carefully: already choosing $\omega \geq 1.4$ leads to divergence of all test cases. For matrices with considerable off-diagonal parts, using the block-asynchronous iteration may not pay off when comparing with SOR. Considering the runtime analysis for the matrix CHEM97ZTZ (Figure 4a) we have to realize that although the unweighted block-asynchronous iteration generates the solution faster than SOR, using ω-weights is not beneficial. The algorithm also does not benefit from enhancing it by ℓ_1-weights, which may stem from the very unique matrix properties. We notice however that, despite the poor performance, ℓ_1-weights have positive impact on the algorithm's stability: for $\omega = 1.1$, the convergence of the block-asynchronous iteration is maintained. For the test matrix TREFETHEN_2000, the performance of SOR and block-asynchronous iteration is comparable for ω near 1. But enhancing the latter one with ℓ_1 weights causes significant performance increase for async-(5). We then outperform the SOR algorithm by a factor of nearly 5 (see Figure 4d). This not only reveals that using ℓ_1-weights is beneficial to the method's performance, but also the potential of applying a combination of both weighting techniques.

4 Conclusion

We introduced two weighting techniques for block-asynchronous iteration methods that improve the convergence properties. In numerical experiments with linear systems of equations taken from the University of Florida Matrix collection we were able to show how the different techniques improve not only the convergence rate but also the time-to-solution performance. The further research in this field will focus on how these improvements by using weights in block-asynchronous methods transfer to multigrid methods. Especially algebraic multigrid, where considerable off-diagonal entries are expected on the different grid levels, may benefit from using weighted block-asynchronous iteration smoothers.

References

[AD86] Aydin, U., Dubois, M.: Generalized asynchronous iterations, pp. 272–278 (1986)

[AD89] Aydin, U., Dubois, M.: Sufficient conditions for the convergence of asynchronous iterations. Parallel Computing 10(1), 83–92 (1989)

[ATDH11] Anzt, H., Tomov, S., Dongarra, J., Heuveline, V.: A block-asynchronous relaxation method for graphics processing units. Technical report, Innovative Computing Laboratory, University of Tennessee, UT-CS-11-687 (2011)

[ATG+11] Anzt, H., Tomov, S., Gates, M., Dongarra, J., Heuveline, V.: Block-asynchronous Multigrid Smoothers for GPU-accelerated Systems. Technical report, Innovative Computing Laboratory, University of Tennessee, UT-CS-11-689 (2011)

[Bag95] Bagnara, R.: A unified proof for the convergence of jacobi and gauss-seidel methods. SIAM Rev. 37, 93–97 (1995)

[Bah97] Miellou, J.C., Rhofir, K., Bahi, J.: Asynchronous multisplitting methods for nonlinear fixed point problems. Numerical Algorithms 15(3-4), 315–345 (1997), cited By (since 1996) 23

[BE86] Bertsekas, D.P., Eckstein, J.: Distributed asynchronous relaxation methods for linear network flow problems. In: Proceedings of IFAC 1987 (1986)

[BFG+] Baker, A.H., Falgout, R.D., Gamblin, T., Kolev, T.V., Martin, S., Yang, U.M.: Scaling algebraic multigrid solvers: On the road to exascale. In: Proceedings of Competence in High Performance Computing, CiHPC 2010 (2010)

[BFKMY11] Baker, A.H., Falgout, R.D., Kolev, T.V., Yang, U.M.: Multigrid smoothers for ultra-parallel computing, LLNL-JRNL-435315 (2011)

[BMPS99] Bai, Z.-Z., Migallón, V., Penadés, J., Szyld, D.B.: Block and asynchronous two-stage methods for mildly nonlinear systems. Numerische Mathematik 82, 1–20 (1999)

[BSS99] Blathras, K., Szyld, D.B., Shi, Y.: Timing models and local stopping criteria for asynchronous iterative algorithms. Journal of Parallel and Distributed Computing 58, 446–465 (1999)

[CM69] Chazan, D., Miranker, W.: Chaotic Relaxation. Linear Algebra and Its Applications 2(7), 199–222 (1969)

[DB91] Dubois, M., Briggs, F.A.: The run-time efficiency of parallel asynchronous algorithms. IEEE Trans. Computers 40(11), 1260–1266 (1991)

[FS00] Frommer, A., Szyld, D.B.: On asynchronous iterations. Journal of Computational and Applied Mathematics 123, 201–216 (2000)

[int] Intel C++ Compiler Options. Intel Corporation. Document Number: 307776-002US

[NVI09] NVIDIA Corporation. NVIDIA CUDA Compute Unified Device Architecture Programming Guide, 2.3.1 edn. (August 2009)

[NVI11] NVIDIA Corporation. CUDA TOOLKIT 4.0 READINESS FOR CUDA APPLICATIONS, 4.0 edn. (March 2011)

[Saa03] Saad, Y.: Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, Philadelphia (2003)

[Str97] Strikwerda, J.C.: A convergence theorem for chaotic asynchronous relaxation. Linear Algebra and its Applications 253(1-3), 15–24 (1997)

[Var10] Varga, R.S.: Matrix Iterative Analysis. Springer Series in Computational Mathematics. Springer (2010)